C++ Programming

Lecture 7
Software Engineering Group

Philipp D. Schubert
Contents

1. Template metaprogramming
2. Variadic template arguments
3. Smart pointer
Template metaprogramming

- Template metaprogramming is a Turing complete language
  - Every intuitive computable number can be computed
    - Meaning: we can basically compute anything
  - Funny implication
    - There cannot be a correct C++ compiler!
- TMP is a bit esoteric
  - Many software companies do not allow it
- Try to use constexpr (since C++11) instead of TMP
  - You will see why that is!
Template metaprogramming prerequisites

- static variables in struct/class
  - Are shared across all variables of that type
  - Belong to the type itself
- Great news
  - Types can store values
    - And with values we can perform computations
    - So we can perform computations with types
  - Templates are processing types!
- TMP uses types in order to express computations

```cpp
#include <iostream>
using namespace std;

struct A {
  // `value` exists only once across all variables of type A
  static const int value = 100;
};

int main() {
  A a, b;
  cout << a.value << 'n';
  cout << b.value << 'n';
  return 0;
}
```
Template metaprogramming

- Functional language
  - Compute using recursion
- Example compute the power function

```cpp
#include <iostream>
using namespace std;

template<int B, unsigned E>
struct power {
    static const int value = B *
        power<B, E - 1>::value;
};

template<int B>
struct power<B, 0> {
    static const int value = 1;
};

int main() {
    const int p = power<2, 10>::value;
    cout << p << '
';
    return 0;
}
```
Template metaprogramming

```
#include <iostream>
using namespace std;

template<int B, unsigned E>
struct power {
    static const int value = B *
        power<B, E - 1>::value;
};

template<int B>
struct power<B, 0> {
    static const int value = 1;
};

int main() {
    const int p = power<2, 10>::value;
    cout << p << 'n';
    return 0;
}
```

- In programming using templates
  - Types are used as functions
  - They can get
    1. Types
    2. Constant values
    3. References to functions
   - as input parameters
   - They can store a
     1. type with typedef
     2. constant with enum or static const
- Template specialization directs control flow
- In our example
  - Template gets instantiated …
  - until the base case is reached
# Template metaprogramming

```cpp
#include <iostream>
using namespace std;

template<int B, unsigned E>
struct power {
    static const int value = B *
        power<B, E - 1>::value;
};

template<int B>
struct power<B, 0> {
    static const int value = 1;
};

int main() {
    const int p = power<2, 10>::value;
    cout << p << '
';
    return 0;
}
```

```cpp
#include <iostream>
using namespace std;

constexpr int power(int base, unsigned exp) {
    return (exp == 0) ? 1
        : base*power(base, exp - 1);
}

int main {
    constexpr p = power<2, 10>;  
    cout << p << '
';
    return 0;
}
```
Template metaprogramming

- Even data structures can be realized
- Remember triple from the exercises
- C++ tuple data type is implemented using template metaprogramming
- Lists are also possible
Computing Eulers number at compile time

- Use this formula for $e$

$$
e = 1 + \frac{1}{1} + \frac{1}{1 \cdot 2} + \frac{1}{1 \cdot 2 \cdot 3} + \frac{1}{1 \cdot 2 \cdot 3 \cdot 4} + \cdots
$$

$$
= \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots
$$

$$
= \sum_{k=0}^{\infty} \frac{1}{k!}
$$
#include <iostream>
using namespace std;

template<int N, int D>
struct Frac {
    const static int Num = N;
    const static int Den = D;
};

template<int X, typename F>
struct Mult {
    typedef Frac<X*F::Num, X*F::Den> value;
};

template<int X, int Y>
struct GCD {
    const static int value = GCD<Y, X % Y>::value;
};

template<int X>
struct GCD<X, 0> {
    const static int value = X;
};

template<typename F>
struct Simplify {
    const static int gcd = GCD<F::Num, F::Den>::value;
    typedef Frac<F::Num / gcd, F::Den / gcd> value;
};

template<typename X1, typename Y1>
struct SameBase {
    typedef typename Mult<Y1::Den, X1>::value X;
    typedef typename Mult<X1::Den, Y1>::value Y;
};

template<typename X, typename Y>
struct Sum {
    typedef SameBase<X, Y> B;
    const static int Num = B::X::Num + B::Y::Num;
    const static int Den = B::Y::Den;
    typedef typename Simplify<Frac<Num, Den>>::value value;
};
Computing Eulers number at compile time (TMP) II

```
template<int N>
struct Fact {
    const static int value = N*Fact<N-1>::value;
};

template<>
struct Fact<0> {
    const static int value = 1;
};

template<int N>
struct E {
    const static int Den = Fact<N>::value;
    typedef Frac<1, Den> term;
    typedef typename E<N-1>::value next_term;
    typedef typename Sum<term, next_term>::value value;
};

template<>
struct E<0> {
    typedef Frac<1, 1> value;
};
```

```
int main() {
    typedef E<12>::value X;
    cout << "e = " << (1.0 * X::Num / X::Den) << '\n';
    cout << "e = " << X::Num << " / " << X::Den << '\n';
    return 0;
}
```

[Example taken from https://monoinfinito.wordpress.com/series/introduction-to-c-template-metaprogramming/]
Computing Euler’s number at compile time (**constexpr**) III

- Using the same formula

```cpp
#include <iostream>
using namespace std;

constexpr unsigned factorial(unsigned n) {
    return (n == 0) ? 1 : n * factorial(n - 1);
}

constexpr double euler(unsigned n) {
    double e = 1;
    for (unsigned i = 1; i <= n; ++i) {
        e += 1.0 / factorial(i);
    }
    return e;
}

int main() {
    constexpr double e = euler(12);
    cout << "Euler’s number is: " << e << '\n';
    return 0;
}
```

- Let’s see what the compiler does

- Compile with:

  ```bash
  clang++ -std=c++14 -Wall -emit-llvm -S euler.cpp
  ```

  (obtain compilers internal representation)
Pros & cons using template metaprogramming

- **Pros**
  - Evaluated at compile time
  - Higher abstraction possible

- **Cons**
  - Compile time gets longer
  - Hard to read/ write
  - Functional style does not match C++
  - Not supported by development tools
  - Error messages make no sense at all
  - It is heavily overused

- **Use C++ constexpr instead!**
- Unless you really know what you are doing
Variadic template arguments

Example: add function from exercises

```cpp
#include <iostream>
using namespace std;
template<class T>
T add(T t) {
    return t;
}
template<class T, class... Args>
T add(T t, Args... args) {
    return t + add(args...);
}

int main() {
    int sum = add(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    cout « sum « 'n';
    return 0;
}
```

Compiler can oftentimes deduce template parameter(s)

Variadic template arguments

Another example: printing everything

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    print_everything("Hello",
                     1,
                     2.333,
                     string("Welt"));
    return 0;
}

// Another example: printing everything

template<class T>
void print_everything(T t) {
    cout << t << '\n';
}

template<class T, class... Args>
void print_everything(T t, Args... args) {
    cout << t << " ";
    print_everything(args...);
}
```

[Have a look at http://eli.thegreenplace.net/2014/variadic-templates-in-c/]
Smart pointer

- Remember (raw) pointers
  
  ```cpp
  int i = 42;
  int *i_ptr = &i;
  ```

- Pointers are necessary for dynamically memory allocation (heap)
  
  ```cpp
  int *dyn_array = new int[12];
  delete[] dyn_array;

  int *dyn_int = new int;
  delete dyn_int;
  ```

- What was the problem here?
  
  - You probably forget to `delete` at some point
  - Finding memory leaks can costs days/ weeks/ $\infty$ amount of time

- Smart pointer (SP) provide safe wrapper classes for raw pointers
Ownership problematic

```cpp
matrix* matrix_multiply(matrix* a, matrix* b) {
    matrix c = new matrix(a.rows(), b.cols());
    // perform the computation c = a * b;
    return c;
}
```

- **Problem**
  - Who frees `c`, allocated in `matrix_multiply()`?
  - It has to be deleted at some point
- **Problem in general:** Who is responsible, who owns the resources?
  - Who allocates memory and who frees it after usage?
    1. Caller allocates, caller frees (see right)
    2. Callee allocates, caller frees (see above)
    3. Callee allocates, callee frees (see `std::string`, `std::vector`)
Smart pointer

- Help with ownership problematic
  - SP know who owns what resources
- SP do the clean-up (`delete`) themselves
  - Call destructor if they do not have an owner anymore
    - Meaning are no longer used by anyone
  - How?
    - SP calls `delete` for object pointing-to when their own deconstructor is called
    - Smart pointer know about ownership!

- This is not a real garbage collector
- This is just reference counting
  - “Only pay for counter-variables & incrementing/ decrementing counters”
- By the way: it is possible to leak resources in Java (although it has a garbage collector)
Smart pointer

- Three types of smart pointers exist
  - `unique_ptr` // for unique ownership
    - One user at a time
  - `shared_ptr` // for shared ownership
    - One or more users at a time
  - `weak_ptr` // for non-owned things
    - Does not own, but is allowed to use the underlying object

- SP are implemented in STL
- All defined in `<memory>`
  - Use `#include <memory>`
unique_ptr

- unique_ptr behaves like a usual pointer
- Example

```cpp
struct Data {
    double x;
    double y;
    Data(double x, double y) : x(x), y(y) {} }
};

int main() {
    unique_ptr<Data> data_ptr(new Data(12.5, 14.8));
    return 0;
}
```

- Notice we do not use `delete`

Did it work?

```
It worked! Memcheck, a memory error detector
Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
Using Valgrind-3.10.1 and LibVEX; rerun with --show-references for copyright info
Command: ./unique

HEAP SUMMARY:
in use at exit: 0 bytes in 0 blocks
total heap usage: 1 allocs, 1 frees, 16 bytes allocated
All heap blocks were freed -- no leaks are possible
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

GREAT!
unique_ptr

- Using the factory function

```cpp
struct Data {
    double x;
    double y;
    Data(double x, double y) : x(x), y(y) {}
};

int main() {
    unique_ptr<Data> data_ptr(make_unique<Data>(12.5, 14.8)); // use make_unique
    return 0;
}
```

- Caution: `make_unique()` exists since C++14
  - It has been ‘kind of’ forgotten in C++11
  - In C++11 just use `new`
**unique_ptr**

1. How to model a `unique_ptr`?
   - Make it a class providing a pointer to a resource

2. How to ensure `data_ptr` is the only user?
   - Prohibit copying it
     ```cpp
     unique_ptr(const unique_ptr& up) = delete;
     unique_ptr& operator= (const unique_ptr& up) = delete;
     ```
   - Now we can only have one `data_ptr`
   - Attempts of copying result in an compiler error

3. How is `data_ptr` able to delete its resource?
   - Use it´s destructor
     ```cpp
     ~unique_ptr() { delete resource; }  
     ```
   - Now the resource is cleaned up for us

4. How to use it elsewhere without copying?
   - Use `std::move()`

```cpp
struct Data {
    double x;
    double y;
    Data(double x, double y) : x(x), y(y) {}  
};

int main() {
    unique_ptr<Data> data_ptr(make_unique<Data>(12.5, 14.8));
    return 0;
}
```
Let’s implement our own `unique_ptr` on the board

Example
This code does not compile

Why?
- `unique_ptr` cannot be copied
- Because copying results in more than one user!

Here we would have two owners
- `main()`
- `setZero()`

Move data instead of copying to have one user at a time
- `move()` `data_ptr` into `setZero()`
- and back from `setZero()` to `main()`

---

**Example**

```cpp
struct Data {
    double x;
    double y;
    Data(double x, double y) : x(x), y(y) {};
};

unique_ptr<Data> setZero(unique_ptr<Data> d) {
    d->x = 0.0;
    d->y = 0.0;
    return d;
}

int main() {
    unique_ptr<Data> data_ptr(new Data(12.5, 14.8));
    unique_ptr<Data> zero = setZero(data_ptr);
    cout << zero->x << 'n';
    cout << zero->y << 'n';
    return 0;
}
```
**unique_ptr**

- **Example**

```cpp
struct Data {
    double x;
    double y;
    Data(double x, double y) : x(x), y(y) { }
};

unique_ptr<Data> setZero(unique_ptr<Data> d) {
    d->x = 0.0;
    d->y = 0.0;
    return d;
}

int main() {
    unique_ptr<Data> data_ptr(new Data(12.5, 14.8));
    unique_ptr<Data> zero = setZero(move(data_ptr));
    cout << zero->x << '\n';
    cout << zero->y << '\n';
    return 0;
}
```

- **This works**
- **Caution!**
  - You better not use `data_ptr` after you moved it somewhere else!
    - Undefined behavior
    - Segmentation fault
  
- **The second move() is “hidden”**
  - `setZero()` moves `d` back to `main()` into the variable `zero`
  
- **Compiler complains if you forget move()**
  - Do not worry
**shared_ptr**

- Allows multiple owners
- **Example**

```cpp
struct Data {
    double x; double y;
    Data(double x, double y) : x(x), y(y) {};
};

shared_ptr>Data> setZero(shared_ptr<Data> d) {
    d->x = 0.0;
    d->y = 0.0;
    return d;
}

int main() {
    shared_ptr<Data> data_ptr(new Data(12.5, 14.8));
    shared_ptr<Data> zero = setZero(data_ptr);
    cout << zero->x << '\n';
    cout << zero->y << '\n';
    return 0;
}
```

- Keeps track of is owners via internal counter
- setZero() can now be used without move()
- It can be copied
- We allow more than one user!
- Does it still clean-up?
shared_ptr

- Improved example

```cpp
struct Data {
    double x; double y;
    Data(double x, double y) : x(x), y(y) { }
};

shared_ptr<Data> setZero(shared_ptr<Data> d) {
    d->x = 0.0;
    d->y = 0.0;
    return d;
}

int main() {
    shared_ptr<Data> data_ptr(make_shared<Data>(12.5, 14.8));
    shared_ptr<Data> zero = setZero(data_ptr);
    cout << zero->x << '
';
    cout << zero->y << '
';
    return 0;
}
```

- make_shared() makes a difference
  - Does only one allocation for data and reference counter
  - data and reference counter sit in one block of memory
  - More efficient!
**shared_ptr**

1. **How to model a `shared_ptr`?**
   - Make it a class providing a pointer to a resource

2. **How to store the references?**
   - Store them in a counter

3. **How to copy?**
   - Just perform a **flat copy** of the handle (do not copy resource)
   - Increment the reference counter

4. **When to delete the resource?**
   - `~shared_ptr`
     ```cpp
     if (--refcounter == 0) delete resource;
     ```

---

**struct Data**
```cpp
data
struct Data {
    double x; double y;
    Data(double x, double y) : x(x), y(y) { }
};
data
shared_ptr<Data> setZero(shared_ptr<Data> d) {
    d->x = 0.0;
    d->y = 0.0;
    return d;
}
data
int main() {
    shared_ptr<Data> data_ptr(make_shared<Data>(12.5, 14.8));
    shared_ptr<Data> zero = setZero(data_ptr);
    cout << zero->x << '
';
    cout << zero->y << '
';
    return 0;
}
data
```
**weak_ptr**

- Can hold a reference but is not an owner

```cpp
#include <iostream>
#include <memory>

using namespace std;

weak_ptr<int> wp;

void f(){
    if (shared_ptr<int> spt = wp.lock())
        cout << *spt << '\n';
    else
        cout << "wp is expired" << '\n';
}

int main() {
    auto sp = make_shared<int>(42);
    wp = sp;
    f();
}
```

- You rarely use it
- But it has its use
- A **weak_ptr** must be copied into a **shared_ptr** in order to use it
A note on smart pointer

- You will love them
- Memory leaks will not happen anymore
- Always prefer using smart pointers when (managing resources)
- If it makes sense, prefer `unique_ptr` over `shared_ptr`
- Smart pointer behave like raw pointer
  - Need just a tiny bit more memory
- Only fallback to raw pointers …
  - if you cannot afford a few bytes more per variable
  - if your platform does not provide a STL implementation
  - if you implement algorithms
A note on dynamic memory allocation

- If you have to dynamically allocate objects
  - Use smart pointer
- If you have to dynamically allocate an array of objects
  - Use vector

- Do not think there are no exceptions
  - Raw pointers are still needed
    - When implementing algorithms
    - If you are only a user and not an owner of a resource
    - ...

Status Quo

- You know very much about modern C++
  - Probably more than your older professors

- What is next?
  - We have to deepen your knowledge!
  - There will be a Christmas exercise sheet with 16 additional points
  - Object oriented programming (OOP)
  - Threads and asynchronous tasks (running computations in parallel)
  - High performance computing (HPC) and what you should know about it
  - Static analysis (SA) and job offers
  - Introduction to the final project as well as hacks and miscellaneous

- A nice talk by Bjarne Stroustrup recaps everything so far and more:
  - https://www.youtube.com/watch?v=86xWVb4XlyE
Recap

- Template metaprogramming
- Variadic template arguments
- Ownership
- Smart pointer
  - unique_ptr
  - shared_ptr
  - weak_ptr
- Status quo
Thank you for your attention

Questions?